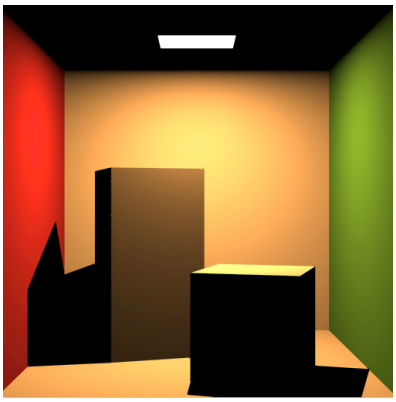
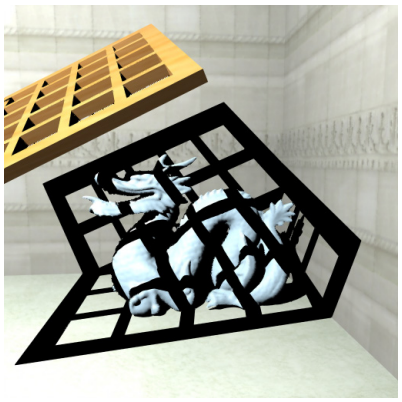
Cornell University
Computer Science Department

Implementing the Render Cache and the Edge-and-Point Image On Graphics Hardware


Edgar Velázquez-Armendáriz, Eugene Lee, Bruce Walter, Kavita Bala

Technical Report

Abstract

The render cache and the edge-and-point image (EPI) are techniques that permit high quality rendering at interactive rates of models illuminated with complex ray traced techniques, combining sparse sampling and discontinuities-respecting interpolation. The image reconstruction is decoupled from the samples generation process and permits the use of arbitrary shaders to gather shading samples. Although the system uses seemingly familiar graphics operations, their behavior differ in subtle and interesting ways from the regular graphics hardware use.

This work presents a multi-pass rendering algorithm that brings the render cache and EPI image generation processes to programmable graphics hardware utilizing their newest capabilities. Its implementation permits substantial performance gains and leverages the CPU workload, allowing more time to be spent on the samples generation. It discusses the performance achieved, optimizations and limitations with the current generation hardware as well as possibilities for future improvements.

November 27th 2005.

Index

1. Introduction

Rendering using graphics hardware has been constrained to use simplified illumination techniques such as Gouraud or Phong shading due to the inherent complexity of ray-traced methods, which are prohibitively expensive in an interactive context. Hardware based ray-tracing [4] and photon mapping [5] are topics of current research interest, however they remain too slow for true interactive rendering. Usage of this kind of computational expensive techniques remain largely absent from real time applications, leaving global illumination and photon mapping mostly for high quality non-interactive renderers.

As an alternative the render cache [12][13] fills this gap between high quality rendering and interactive contexts. It relies upon an underlying renderer and the communication between both elements is asynchronous, enabling the frame rate to be independent from the renderer's speed, employing colored sample points that are reused between frames as the display primitive. Because the straightforward image reconstruction from those sparse samples produces objectionable blurring, Bala et al. [1] proposed the edge-and-point image (EPI), which reconstructs the images respecting important visual discontinuities like shadow and silhouette edges. The explicit discontinuities representation permits the creation of anti-aliased images without supersampling.

In this report we present a multi-pass algorithm for the GPU that builds on the render cache and the EPI approach to create a renderer targeted at the interactive use of global illumination and related shaders on regular desktop systems with commodity graphics hardware. We present a hybrid system where the GPU is used for the complete image generation process as well as the detection of silhouette edges; the CPU work is limited to the communication with the asynchronous renderer. We found that this approach is not only faster than the previous software only method but also allows the CPU to use more time in other tasks like calculating shading samples, leading to a faster image convergence.

1.1. Related work

There has been considerable recent interest in display processes that can automatically exploit spatial and temporal coherence to allow interactive use of shading algorithms that would otherwise be too slow for interactive use. Some examples also use graphics hardware like the shading cache [11], corrective texturing [9] and trapestry [7]. In a non-interactive context, the irradiance caching [8] stores irradiance samples sparsely on the objects to speed up the shading of dynamic scenes that use global illumination.

Pighin [3] explored using edges to improve image reconstruction from sparse points for progressive update of a still image. The edge-and-point renderer [1] and the silhouette shadow maps [6] each explore image representations that combine sharp discontinuities with sparse samples for different goals. Silhouette shadow maps embed approximate edges to reduce artifacts from limited resolution shadow maps. However the edge-and-point system was the first to demonstrate that combining explicit edges and sparse point is feasible in an interactive context.

Another promising approach is to accelerate traditionally slow shading algorithms by mapping them to graphics hardware, for example, implementing completely ray tracing [4] or photon mapping [5] onto programmable GPUs. Szirmay-Kalos et al. presented approximate ray tracing using distance impostors [10] as a way to generate estimated reflections, refractions and caustics using specially addressed environmental maps. Since these can be used to produce shading samples for the display process, we view these approaches as orthogonal and complementary to the work presented here.

2. Algorithm overview

We will proportionate a brief overview of the core algorithms underlying this system, the render cache [12][13] and the edge-and-point image [1]. Both of them work together to create the final images.

2.1. Points projection

The render cache works by caching rendering results provided by an arbitrary shader and reusing them along several frames, using the stored points to estimate the current image. The points are saved as 3D points with an associated color, an identifier of the pixel to which they were projected on the previous frame and an age in a fixed-size structure called the point cloud. The render cache projects the point cloud to the image plane using z-buffering from the new camera position, recording on the projected pixel images the color, sub-pixel location information and the identifier of the point that was projected to that pixel. As there is normally not a one-to-one mapping between points and image's pixels, it applies some filters to correct disparities in the projected data. A depth cull procedure removes points that are likely to be behind foreground surfaces and creates an interpolated depth image, which is constructed by reading the depth information from nearby pixels and moving it back by a threshold related to the scene dimension (we are currently using 5%). A final image is reconstructed from the remaining samples using different interpolation/smoothing filters.

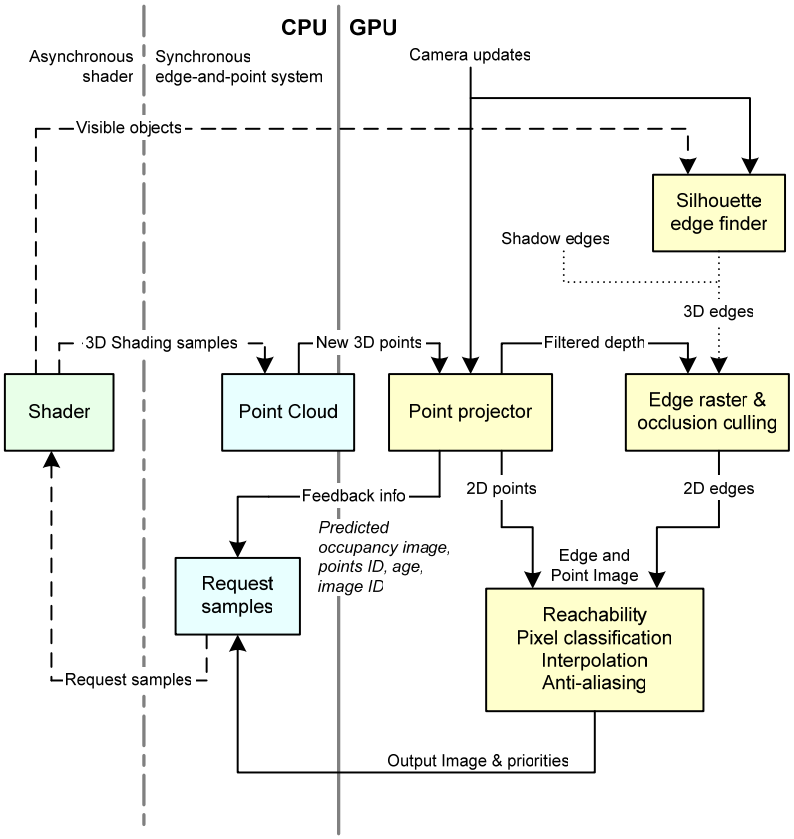


Figure 1: Overview of the GPU system architecture.

2.2. Samples request

The system expects that the renderer can generate only a small number of new samples every frame, therefore is very important to guide their location to obtain the maximum benefit. The render cache directs the renderer to focus on regions determined by the means of a predicted occupancy image, which contains the projection of the cloud's points from an estimate future position, and by receiving feedback from the interpolation process, using an error-driven dither to select the best locations for sampling. The new samples are integrated to the point cloud as they become available replacing the eldest points.

Old points are automatically evicted after they reach certain age even if no point has replaced them. The points may be further penalized if they were projected outside the view frustum or if considerable color change has occurred nearby the projected point's position. This prevents stalls and leads to faster image convergence.

2.3. Edges processing

The EPI is a method to approximate a final image from sparse samples interpolating them only between regions that are not separated by important discontinuities. The system tracks shadows boundary and silhouette edges. The shadow edges are view-independent and are updated only when it is needed. On the other hand the silhouette edges must be computed every time the viewing position changes. The detected edges are rasterized to the image plane flagging each intersection with the pixels boundaries at one eight of pixel precision. As the pixels share boundaries, they store only the intersections on two of their sides, the other intersections may be read by looking at the pixel's neighbors.

2.4. Pixel classification

The system classifies each pixel either as empty, simple or complex based on the edge crossings recorded on its boundaries. Empty pixels have no crossings, simple pixels have exactly two intersections and all other instances are classified as complex.

Simple pixels contain just one approximated edge even if many small ones cross the pixel. This edge divides the pixel into a primary and a secondary region. The empty and complex pixels contain only a primary region. The interpolation stage computes only the color for the pixel's primary region, whereas the secondary region's color is estimated during anti-aliasing.

For simple pixels that contain a color sample we need to determine in which side of the edge that samples lies. Because of the limited precision of the location of the sample and the edge position we may not be able to resolve this reliably. Such samples are named ambiguous and we invalidate them to prevent their further use, otherwise the side containing the sample is considered the primary.

2.5. Reachability

Edge-respecting interpolation relies on the concept of reachability. A sample is reachable from a region if a direct path connecting both of them that does not cross any boundary exists; complex pixels are always considered unreachable. Reachability is propagated from each pixel to its 3x3 neighborhood using three basic operations: neighbor reachability, chaining and combining, and it is encoded in a four bit mask.

The neighbor reachability finds the mask between neighboring pixels based on their edge classification. Chaining combines the reachability masks from a source pixel to an intermediate pixel and vice versa into a reachability mask from source to destination. Lastly we combine the

reachability through intermediate pixels. Because the reachability is reflexive we can save some processing time by reusing the values from pixels in previous scanlines.

2.6. Interpolation

Interpolation combines all the valid reachable samples in the pixels' 5x5 neighborhood using a center weighted kernel. Missing or unreachable samples receive a weight of zero and the weights are renormalized for each pixel. Complex pixels contain an edge configuration that is too complicated to reconstruct accurately, so they are handled by a regular interpolation that ignores reachability. Although this interpolation will be less accurate, complex pixels tend to be rare and are located in complicated parts of the image where errors are hard to detect.

This stage calculates the priority value of each pixel, which the dither process will use to request the new samples. For pixels with valid samples their priority is directly related to the age of the point that was projected there, otherwise it is calculated taking into account the number of valid samples around the pixel.

2.7. Anti-aliasing

The interpolation only computes a color for the primary region of the pixels. For simple pixels we also need a color for the pixels' secondary region. The anti-aliasing filter approximates the secondary region's color by replicating the color from the primary region of a neighbor pixel, which is chosen as the one most likely to contain a correct color based on the edge of the pixel analyzed. Finally both colors are blended based on the areas that each region occupies to compute the anti-aliased color.

3. Mapping to the hardware

In this section we will present the way we mapped the combined render cache and EPI system to the GPU, discussing the architecture decisions taken to determine which tasks would remain on the CPU and which ones could be implemented on the hardware. As each process had different challenges, we will also discus the specific performance limitations we encountered.

3.1. Implementation details

The system is divided in three main modules. The main control module is the RTGI system, written in Java, which uses JNI to interface to the native implementation modules. The native modules are written in C++, one contains a modified version of the Render Cache and the other

realizes the GPU functionality, using C++ and Cg for the shaders. Those shaders require at least fp40 support: Shader Model 3.0 fragment programs.

The native graphics hardware module encapsulates all the GPU functionality into classes, each of them manages the creation and destruction of all data required, such as arrays and textures. Also a number of helper C++ classes where created to manage and create OpenGL objects like textures and buffer objects.

The recently available extension EXT_FRAMEBUFFER_OBJECT is used thoroughly: all the off-screen is rendering is done using them. This extension permitted the use of Render To Texture (RTT) and Render To Vertex Array (RTV), both of which were used in this implementation. All the window system dependent code to get a valid OpenGL render context is encapsulated in a particular library class. This is the only platform specific code. The shaders assume that they are being run in a low-endian machine, such as the IA32 architecture.

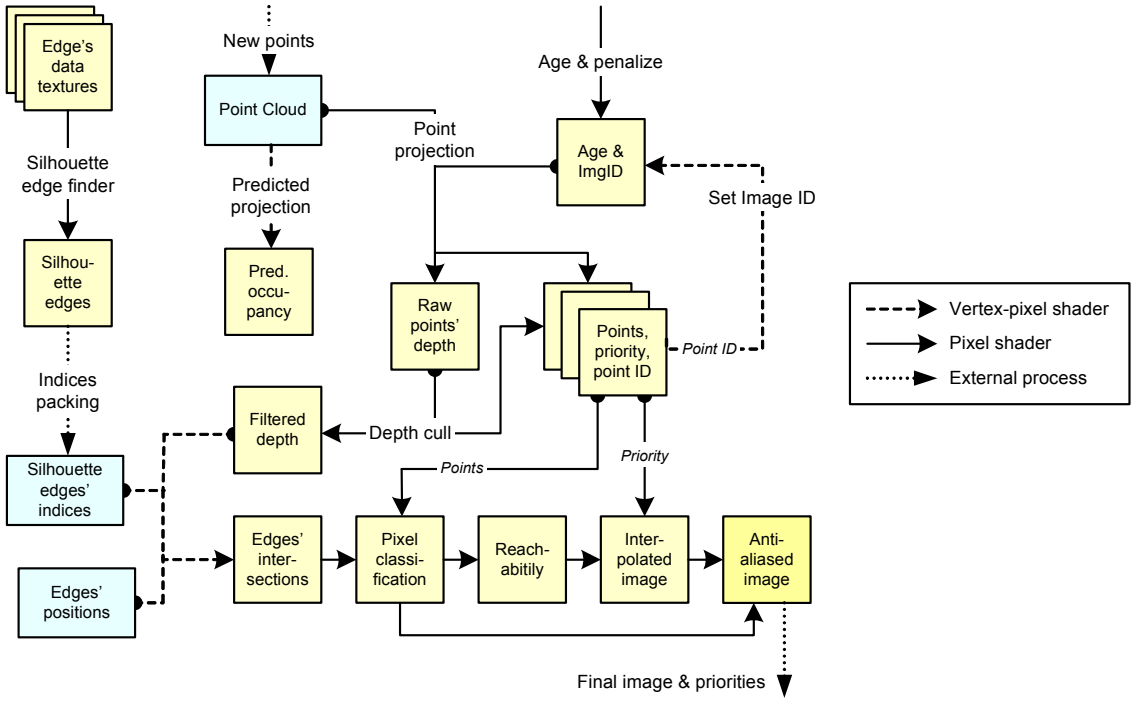


Figure 2: Data flow on the GPU. The squares on the figure represent textures, the rectangles VBOs. Dashed lines are vertex/pixel shaders, solid lines pixel programs and dotted lines processes not handled directly by the GPU.

3.2. Point projection

All drawing operations of the render cache were moved to the GPU. These include the point projection, depth filter and the simple interpolation with adaptive kernels (3x3 of 7x7 in a single pass). Other operations which update intermediate values of the point cloud are also implemented on the GPU such as the points' aging, penalizations assessment and updating the image ID in the cloud.

Projecting the points, using one pixel for each one, involves unsorted writes to the framebuffer. Previous tests shown disappointing results, but with new experiments we realized that the penalty with current graphics hardware is negligible. Reading the assorted vertices in disorder, which is not the case of our system, is slower by an order of magnitude than scattered writes to the framebuffer reading a continuous stream of vertices.

Requesting and updating new samples, as well as registering new ones remain on the CPU. The error based dither image that asks the renderer for new samples is not well suited for GPU computation and it is outside the scope of the project. It also needs to send the information to other processes which may be running on other machines if the shader is distributed. The CPU updates the points with the new samples using a RAM copy of the point cloud; it tone maps the new points and requests penalizations for points that changed notably their color. After the samples have been processed they are uploaded asynchronously to the GPU.

Because the drawing primitives are 3D colored points which are projected onto an image plane the projection process maps straightforward into the GPU's architecture. The best way to send the points' information to the GPU is using a VBO[1], even though its contents will be modified every frame. Even a dynamic VBO is much faster than immediate access mode on a PCIe 16x bus for sending drawing primitives to the GPU. There is also a texture for data of ech point on the GPU, the age (8 bits) and the 24-bit Image ID (the index of the pixel to which this point was drawn in the last frame), or 0x7FFFFF if it was not drawn. The highest bit in the Image ID is used to request a color change penalty for that point during the aging.

Our implementation keeps all the intermediate results as textures, rendering directly to them using FBOs [2]. This way the result of one stage can be fed into upcoming passes without overhead. The image data stored as textures includes the point image, the subpixel information,

[1] VBO: Vertex Buffer Object.

[2] FBO: Framebuffer Object.

the projected points' IDs, priority and the filtered depth. The data from the point cloud that is updated on the GPU, which is also used in during other stages, is also stored as textures.

The total storage needed on the GPU for each point in the point cloud is 24 bytes (3 floating point coordinates, 3 unsigned byte colors, 1 byte flags; also a 24 bit image ID and 1 byte age duplicated).

The point projection writes at the same time various attributes: the point image, subpixel information, partial priority values related to the age of the points and the point's IDs. Therefore we chose to use MRT[3] to project the point cloud only once, writing simultaneously to four textures (three color textures and one depth texture).

The final point image (the plane where the points were projected to, and that will be used for the final image reconstruction through filters) consists of 2 RGBA8 textures. The first one has the color as the RGB components, and the total priority of that pixel in the A channel. The other one is interpreted as a 32 bits unsigned integer, whose uppermost 24 bits are the point ID (the index of the point in the point cloud that mapped to this pixel), and the lowermost 8 bits are the flags. Given the endian-ness of the system architecture used (IA32), in the shaders the MSB of that 32 bit number is stored in the R channel, and the LSB in the A channel. This way the resultant 4 unsigned bytes of each pixel could be stored into a single 32 bits integer without further processing.

During the point projection execution, a vertex shader clamps the depth of all samples to lie at least at the back of the view frustum. If this correction is not applied all the points from the background would be clip-culled and the priority values would be inaccurate as regions with those points would be improperly interpreted as zones with no projected samples. To avoid excessive state changes, we create the predicted occupancy right after the point projection by drawing the point cloud again from a different perspective without depth testing and writing simply non-black pixels on areas with projected points.

This vertex shader also generates subpixel information. Each vertex, which corresponds to a point in the cloud, will map at most to one pixel, and a pair or values (s_x, s_y), each in the range [0,1] is passed to the pixel shader. This value indicates where in the pixel the point was drawn. (0,0) is the top left corner, and (1,1) is the bottom right corner of the pixel. The subpixel information is stored in the Alpha channel of the projected points texture with values in the range

[3] MRT: Multi Render Target.

[0, 15]. Each number corresponds to the index of the pixel's quadrant in which the point sample, according to the information provided by the vertex shader.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 3: Subpixel regions of a pixel.

The depth cull deletes samples from the projection that are likely to be occluded by other surface. A pixel shader computes a filtered depth, which is the average depth of the points that were projected in a 3x3 neighborhood, moved back by a threshold. This stage then erases the pixels that are behind the calculated depth and deletes them simultaneously from the outputs of the point projection using alpha blending and MRT. The edge raster stage of the EPI will use the filtered depth texture created at the depth cull.

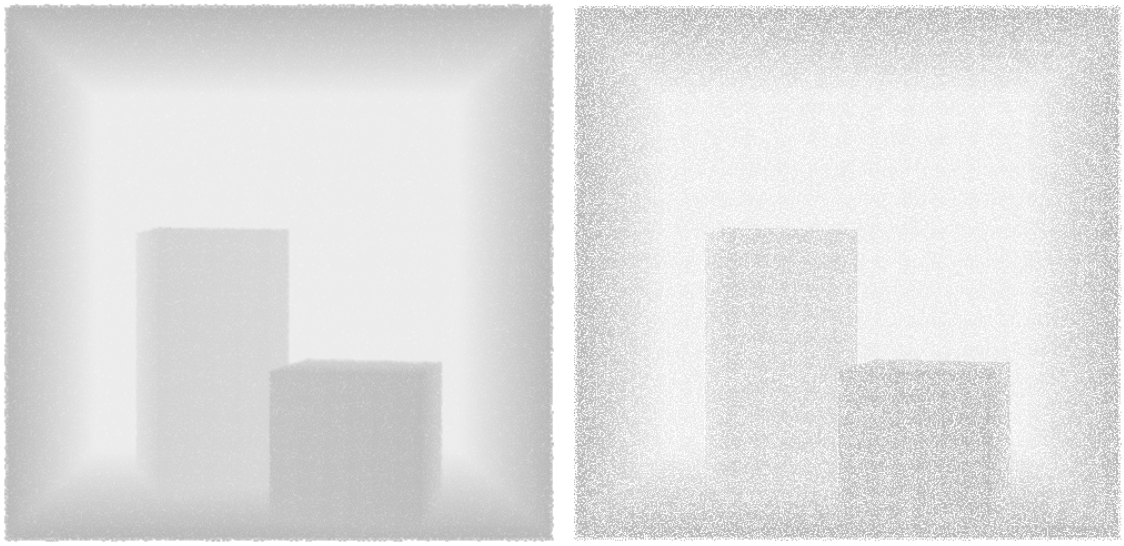


Figure 4: Filtered depth texture (left) and raw projected points' depth (right).

The points' aging and penalization is done on the GPU, with the GPGPU style, processing one point's data per fragment. The current age – image ID texture, with the age reset for the new points and with the penalization flag, is read in a fragment shader. For all points with age greater than zero, the penalizations are verified. An image ID of *0x7FFFFF* means that the point was outside the view frustum and is penalized. This image ID is equivalent to *G = 255, B = 255, A = 127*. Details of this arrangement are provided in the description of the set image ID stage. The

color change penalization flag is verified testing *alpha >= 128/255.0*. The pixel shader cannot make bit operations, and it receives the color components as normalized floating point values. However, for any unsigned byte alpha value a, it is always true that *a | 0x80 >= 0x80 = 128*.The final age is written into the second copy of the texture, and then these textures exchange the source/destination roles.

As an optional component for the system we implemented the interpolation model of the render cache, which consists of a prefilter stage with an uniform 7x7 kernel to fill ample regions without points and a 3x3 weighted interpolation kernel for denser zones. We combined both the prefilter and interpolation stages into a single pass, using Shader Model 3.0 dynamic branching on the pixel shaders. The larger 7x7 prefilter is used only where needed, thus improving the performance.

Limitations

We found that the point projection stage is mainly bandwidth limited. Due to the hybrid nature of our approach we have to read back information from the GPU back to the main memory, this imposes a huge workload on the data bus, and each readback forces the synchronization between the CPU and the GPU. This transfer is the slowest of all, therefore it is very important to avoid unnecessary transfers copying only the data strictly needed. The data transferred is the predicted occupancy image, the projected points' IDs, the points' age and image ID, and the final filtered image with the priorities.

When uploading the new points to the point cloud, only the new ones are written to the VBO, and their age is reset by drawing for each one a single point into the age texture, at the position corresponding to the updated point in the texture. The *i*-th pixel of the texture will be updated if a point is drawn at the position *[(i % texture_length) + 0.5, (i / texture_length) + 0.5]*, which are the coordinates of the center of the *i*-th pixel in the texture. This mapping is precomputed and stored in a static VBO.This technique avoids copying at each frame the complete point cloud and its ages set from the CPU to the GPU. In this same lieu we specify the points to be penalized by color changes by setting a flag in the image ID field.

We found other important limitation on the speed of feeding the point cloud VBO to the vertex pipeline of the GPU. To circumvent this restraint, for the predicted projection only a quarter of the point cloud is projected using splats instead of single pixel points. Alternating between different regions of the point cloud leads to a distributed well enough set of points that yields good results.

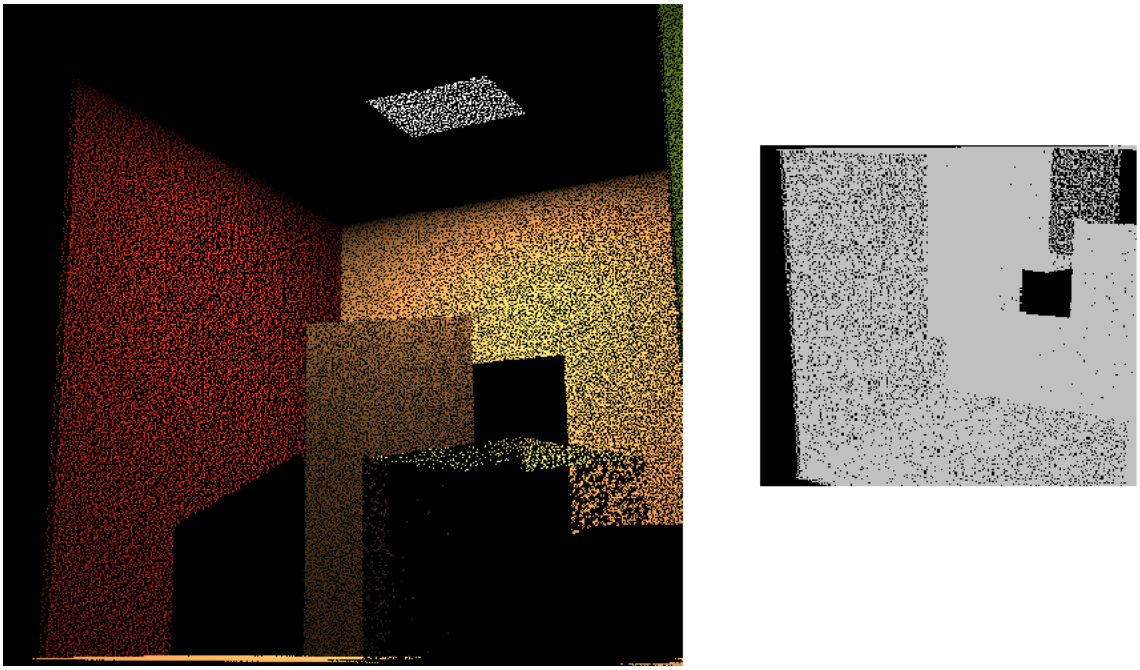


Figure 5: Point projection after the depth cull (left) and the predicted occupancy image (right). The predicted projection image has its y-coordinates inverted to match the way the render cache reads the image and how OpenGL writes it: the render cache begins at the top left corner whereas OpenGL at the bottom left corner. Both transverse each scanline from left to right.

In the original implementation, the render cache writes at each point when it is being projected the ID of the pixel were it was projected. As writing to the incoming vertex data on the GPU is not feasible, an additional pass is used to write the image ID at each member of the point cloud. We achieve this by reusing the point ID projected texture as a vertex buffer using RTV[4]. Each pixel in the texture is translated into a vertex, which will set the image ID for the point that was projected to that pixel. A vertex shader is used to interpret the vertices' positions, which have encoded the ID of the point to update.

3.3. Silhouette detection

The shadow edges are detected on the CPU using a pixel shader, because they require a lot of dynamic information from the scene. The overhead of this design decision is low because once the CPU detects those edges, they will not change as long as the scene does not change; they are independent from the current viewpoint. Hence those edges only need to be transferred to the

[4] RTV: Render to Vertex Array

GPU when they are modified. The system stores the model's edges both at a static VBO, to draw them quickly, and as textures, to detect the silhouette edges with a pixel shader.

The edges are stored in the VBO in interleaved format to achieve better memory coherence and OpenGL performance. For each edge with coordinates (v_n, v_{n+1}), the information stored in the edges array is $(v_{n+1}, v_n, v_n, v_{n+1})$: a pair of a vertex attribute and the position of that vertex. There are four textures, two for the vertices and two for the normals of the edges. Each pixel of every texture holds information for one edge.

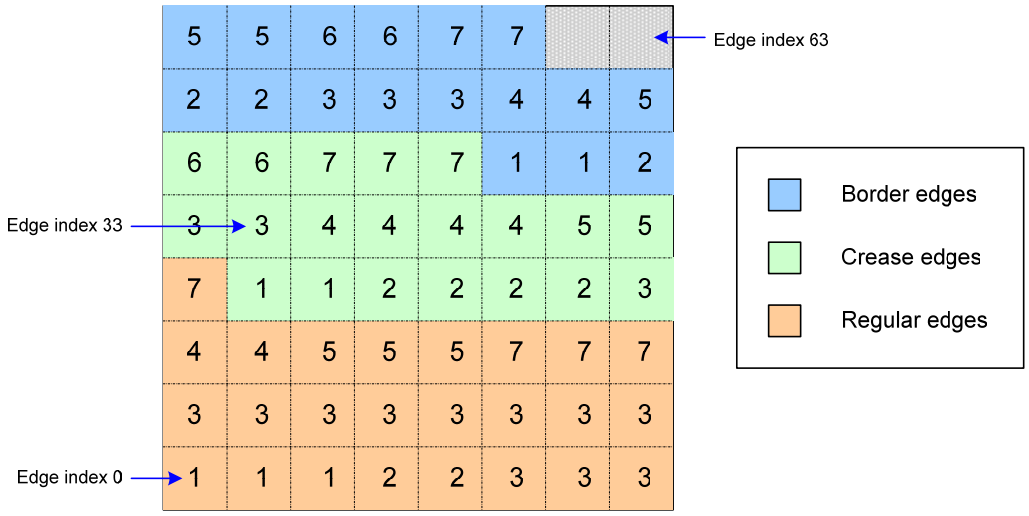


Figure 6: Layout of each of the textures created for the silhouette detection. The number inside each pixel represents the object ID of the object that edge belongs to.

This fragment program writes to a texture that indicates which edges belong to the silhouette. Given the vector with the current eye position v_{eye}, and vector $p = v_0 - v_{eye}$, the test are:

- For regular edges: $sign(p \cdot n_0) \neq sign(p \cdot n_1)$
- For crease edges: $sign(p \cdot n_0) < 0 \vee sign(p \cdot n_1) < 0$
- For border edges: $sign(p \cdot n_0) < 0$
    - For border edges, only n_0 is valid, n_1 is <0,0,0>

Because there is a direct correspondence between the position of each edge's information in the textures and in the VBO, we read this image and fill with the CPU control a tightly packed VBO with the indices of the silhouette edges that will be used along with the edges' static VBO at the edge raster stage. These steps execute at great speed every frame because all data transfers take place on the GPU.

At the results texture if the *i*-th pixel is not black, then the *i*-th edge is in the silhouette. For each silhouette edge, the pair of indices *2i, 2i+1* is written to its destination VBO. These two indices are the positions of the two vertices of that edge in the static data VBO. Because the system stored the first index and count of each edge type for every object, it can read only those edges, skipping those of the non visible objects. And because the set of visible objects is ordered, the indices of the silhouette edges are saved in ascending order.

Limitations

The detection process is a brute-force approach that is limited by the fill rate of the GPU. It is also constrained by the bandwidth requirements of the edges textures, so a simple way to diminish it is using fp16 textures for the normals, which we normalize in order to meet the limited data range. We left the full precision fp32 textures only for the edges vertices positions, since those values might be too big to fit in fp16 numbers. Immediately by implemented this we achieved 20% faster performance.

3.4. Edge raster

After we found the silhouette, the next stage is to raster those discontinuities edges onto the image plane to find subpixel-level intersections with the top and left boundaries of each pixel; the other intersections may be retrieved via the neighbor pixels. o The edges are drawn both from the static model data VBO with the index array VBO packed by the silhouette detector as well as the shadow edges given by RTGI and set up in their own VBO in advance.

To avoid processing the edges occluded by surfaces, the filtered depth from the depth cull is used as a read only depth buffer, comparing the edges' z-values against it. We also cull edges that have a length inferior to one pixel using a vertex shader. We draw the edges with thick lines extending their length by one pixel at each end to ensure that we conservatively active all pixels with potential intersections.

3.5. Image filters

The rest of the stages, pixel classification, reachability, interpolation and anti-aliasing; are implemented as a consecutive set of texture filters and share a similar set of implementation strategies and limitations. These stages use lookup tables encoded as fp16 textures to avoid control code in the fragment shaders, and some of the original operations in the EPI are pre-computed and stored in those textures.

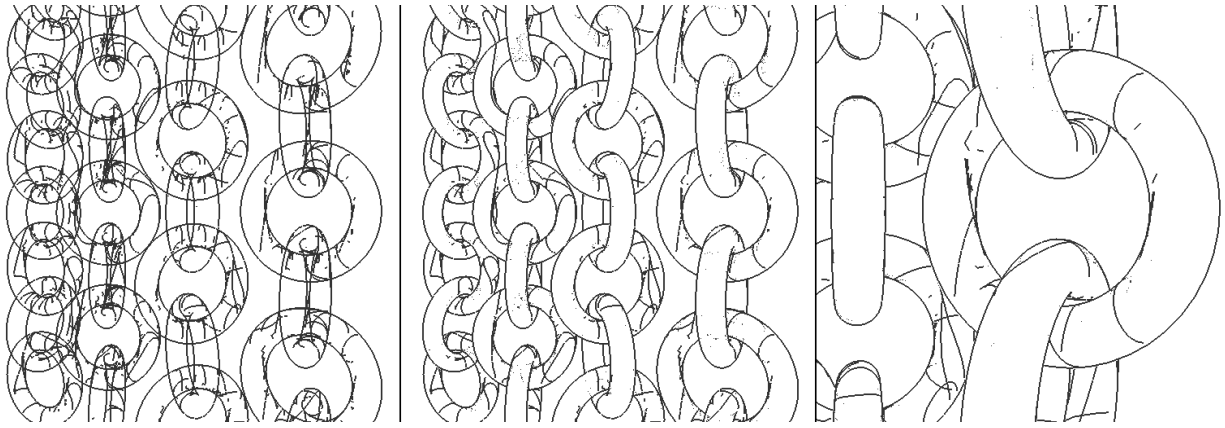


Figure 7: The silhouette detection finds edges that may be behind the surfaces (left). Using the filtered depth texture the edge raster draws the appropriate edges (center). The image on the right shows a detailed view of the edges drawn.

At the classification stage, the pixels are classified as empty, simple and complex depending on the total number of edges' intersections that lay within. Only the simple pixels need comprehensive processing, so dynamic branching is used to reduce the number of pixels totally analyzed.

Due to the limited edge intersection and subpixel information precision some point samples are ambiguous, i.e., it cannot be established in which side of the edge that sample is. Consequently those samples are invalidated from the projected point image, using MRT to make both the classification and the invalidation at the same time, writing the classification to one texture and the valid point samples to other one.

The reachability operation uses three passes, and the interpolation uses that result to blend the samples and calculate the priorities. After the last filter, the anti-aliasing, is applied, the system reads back the final image with the priorities to display it and request new samples.

Branching granularity

The current GPUs have limited performance gains with the dynamic branching on the pixel shaders due to their SIMD nature. The current thread goes through the fast branch only if all the pixels being evaluated take the same path, otherwise all of them follow the long path. Only the regular Render Cache interpolation showed substantial improvements with the branching because almost of the pixels take the shortest path. However other stages such as the pixel classification would require better granularity so that only a small amount of pixels follow the slowest path.

This kind of complex scenes should benefit from newer hardware with a granularity of 4x4 pixels, instead of the 64x64 blocks used with most current generation cards.

Bit operations on the GPU

The current generation of GPUs does not allow bit level operations, but we can still verify if a flag is set by comparing the numerical values. If the k bit of a number n is set, then $n \geq 2^k$.

An effective way to test this is using the step function: $step(a, x) = x \geq a \ ? \ 1 : 0$. This function is better employed in its vector form, comparing four independent values in a single instruction. Using this step as a modulator, the weight and current total of the convolution filters is implemented with neither branching nor predication as:

$$modulator = step(2^k, n) * currentWeight$$
$$total \quad += modulator * currentValue$$
$$weight \quad += modulator$$

To test if only the k bit is set, the floating point operation executed is $step(2^{k-1}, mod(n, 2^k))$.

The convolution filters and other operations require to multiple sums and products in the style $a = c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4$. An effective way to optimize this operation is performing a dot product between the elements of this linear combination: $a = [c_1,c_2,c_3,c_4] \cdot [x_1,x_2,x_3,x_4]$. The dot product of up to four elements vectors is a single hardware instruction.

```
const half4 reachRfmod1 = fmod(reach.rrrr, half4(2,4,8,16));
```

```
const half reachRfmod1a = fmod(reach.r, 2);
const half reachRfmod1b = fmod(reach.r, 4);
const half reachRfmod1c = fmod(reach.r, 8);
const half reachRfmod1d = fmod(reach.r, 16);
```

Figure 8: Vectorized code (top) and regular style code (bottom).

Fill rate and texture access

The EPI filters require abundant texture reads and most of their calculations depend on those values. Therefore faster texture access speeds would benefit those stages. The EPI filters are also limited by the fill rate: the number of pixels the GPU can process. Increasing the pixel pipelines number and clock frequency would decrease the time needed for theses filters.

A technique for reducing the fill rate requirements is to diminish the number of operations in the pixel shader. In all shaders the operations were vectorized whenever possible as a mean to reduce

the instructions count. Moreover the number of registers used by a shader is still important for achieved maximum performance. A 180 instruction shader which uses 25 registers performs 50% slower than other version of the same shader of 215 instructions that uses 24 registers.

Stage	*Instructions*	*R-Regs*	*H-Regs*	*Texture reads*
Point projection	15	1	1	1
Depth cull	38	2	2	9
Age points	17	1	1	1
Silhouette detector	25	3	2	3
Edge raster	23	3	2	0
Pixel class	84	2	3	9
Neighbor reachability	19	2	2	7
Reachability main	140	13	2	40
Copy reachability	66	2	4	10
Interpolation	214	25	9	27
Anti-aliasing	23	3	1	4

Table 1: Pixel shader statistics.

Stage	*Instructions*	*R-Regs*
Point projection	18	2
Depth cull	15	2
Set Image ID	23	2
Edge raster	40	3
Neighbor reachability	8	1
Reachability main	19	2
Copy reachability	17	2

Table 2: Vertex shader statistics.

4. Results

In this section we present detailed results for our hybrid rendering system. All images are 512 x 512 and are rendered with direct illumination with ambient light. Both the shader and the image reconstruction process ran in the same system, a Pentium 4 3.2 GHz dual core with 2 GB of memory. The GPU used is a Nvidia GeForce 7800 GTX with 256 MB of memory, using Forceware drivers version 81.85. The GPU code is written in C++ using OpenGL and Cg 1.4rc. The top level rendering system runs with Java 1.5 and both components are interfaced through JNI[5].

[5] JNI: Java Native Interface

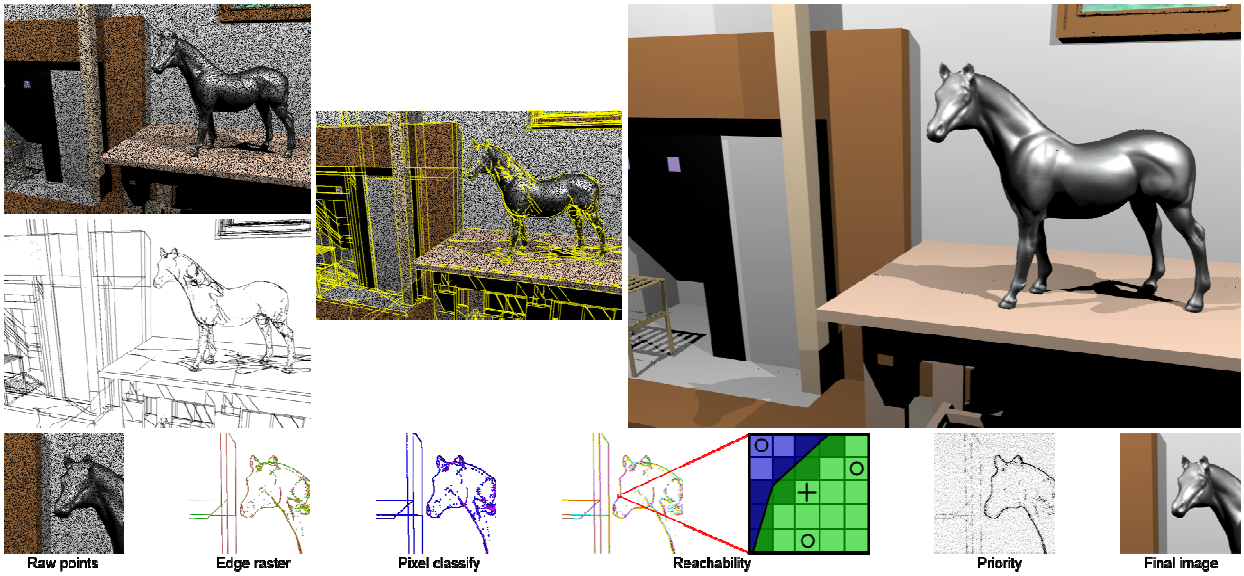


Figure 9: Illustration of the different stages of the system. The points and the edges create the edge-and-point image. Edge constrained interpolation creates a high quality approximation of the image. The bottom images represent details of different stages of the image process.

We present result for five scenes. The Cornell box scene is the simplest one with one area light and 36 polygons. The Chains scene, with two lights, has tessellated non-convex objects (73,728 triangles) casting complex shadows on each other. The Mackintosh room has three lights with 101k triangles. The David head from Stanford's Digital Michelangelo project, has 250k polygons and is lit by 1 light. The Dragon scene, from Georgia Tech's Large Geometry Models Archive, is a 871k polygons model that has a grid casting a shadow pattern on the dragon.

Table 3 shows detailed timings for these models and Table 4 gives a comparison between our implementation and the original render cache and EPI system, run on the same system. The overall frame rate using the GPU is above 60% faster than the original, and the speed up increases along with the scene complexity. On the David head and dragon scenes we achieved twice the previous frame rate. As a side benefit more samples are available each frame for projection. The silhouette detection is also faster using the GPU and depends of the complexity of the scene, as do the edge raster stage. All the other processes are independent from the scene complexity and are related only to the image size; the number of elements in the point cloud depends of the image size as well.

	Original	Full GPU	Speed up
Cornell Box	14.46	23.35	61.46%
Chains	13.71	24.20	76.49%
Mack Room	13.08	21.87	67.15%
David Head	9.25	18.61	101.14%
Dragon	7.42	15.59	110.01%

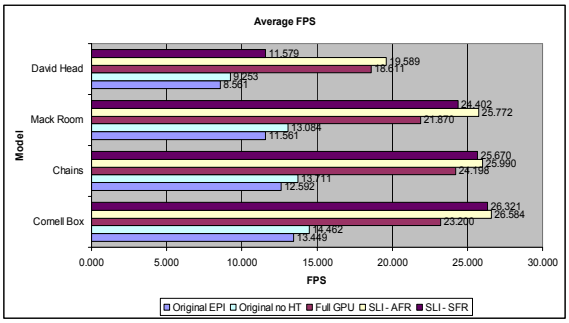


Table 3: Frames per second comparison between the original system and our GPU implementation.

Discussion

Originally we taught that the point projection, being an operation that maps straightforward to the graphics hardware would be the fastest and that the EPI filters, with their huge amount of texture accesses and several passes would be the slowest. Unexpectedly it arose to be the opposite. The graphics cards are designed to draw thousands of pixels from a few vertices and our approach of one pixel per vertex is against that premise. This along with the bandwidth limitations of VBOs cause that the point projection is the new performance bottleneck, taking as much as 33% of the GPU time.

Using a point cloud of roughly 300,000 points, all of them would be projected in 5 ms using a static VBO. Once we updated it every frame, the time needed to stream all the points increased to 11 ms, regardless of the complexity of the vertex and pixel shaders used. A plausible explanation for this conduit is that because the points are updated frequently and the buffer is treated as an entity, they remain in lower performance memory. It is important to recall that when using VBOs the video driver manages memory allocation. During out tests, we found that if the points were updated only each four frames or less frequently, they would be read faster, achieving the original rates of 5 ms. A possible reason is that the driver's memory manager moves the buffer to the fastest memory available after it was used several times without being modified.

To implement the update of the image ID of the point cloud we evaluated to use VTF[6]. However it only supports some texture formats and emerged to be between twice and trice slower than

[6] VTF: Vertex Texture Fetch

RTV, implemented as reading back to a PBO[7] and then using that buffer for feeding the vertex engine.

Implementation specific findings

Rendering to different targets at the same time using MRT is practically free, the overhead is very low. With the current drivers, a FBO could have only attached a depth texture of 24-bit precision, explicitly requested with the internal format `GL_DEPTH_24`. Any other format would yield errors. The function `glDrawRangeElements`, called with the wrong range of indices produces a GL error, but it still renders properly.

Using VTF with a format other than RGB/RGBA 32-bit floating point with target `GL_TEXTURE_2D` trigger software mode, which is about 200 times slower and strange effects occur, such as a missing blue channel. VTF is still slower than other methods, such as RTV.

The Cg compiler excels in instruction scheduling and register allocation, especially for the texture fetch instructions. Very complicated and hardly readable shader code will be compiled into the same program than simpler source file if their underlying structure is the same. Attention must be pay to the output of the Cg compiler when designing the shaders.

Scene	Edges	P_1	P_2	P_3	P_4	E_1	E_2	I_1	I_2	GPU Time	Total Time	FPS
Cornell Box	75	5.14	12.31	1.34	1.08	0.34	0.43	1.09	7.29	32.32	42.82	23.35
Chains	110,592	4.02	10.77	1.30	1.07	1.54	0.99	1.44	7.17	31.81	41.32	24.20
Mack Room	153,526	3.71	9.48	1.20	1.06	1.86	1.93	1.38	7.06	31.17	45.73	21.87
David Head	374,635	4.37	11.92	1.18	1.05	4.92	5.87	1.15	7.11	40.78	53.73	18.61
Dragon	1,305,164	4.15	12.79	1.18	1.06	13.81	7.88	1.25	7.11	52.22	64.16	15.59

Table 4: Performance results. For each scene, the first column gives the number of each model's edges. The next eight columns give the timings in milliseconds for these processes: update points and request samples (P_1), point projection (P_2), predicted projection (P_3), depth cull (P_4), GPU silhouette detection (E_1), edge rasterization (E_2), pixel classification (I_1), reachability, interpolation and anti-aliasing (I_2). The next two columns give total GPU time, including all overheads, and the total frame time (both CPU and GPU) in milliseconds. The last column presents the achieved frames per second.

[7] PBO: Píxel Buffer Object

5. Conclusions

We have presented a GPU-based system built upon the render cache and the edge-and-point image using commodity graphics hardware that is faster than the original implementation. We have shown the different approaches that may be used to map the different stages to the hardware by using it in a non canonical way and overcoming its current limitations.

The system's performance is likely to improve with the current tread of GPUs, which are incorporating more pixel pipelines, higher clock frequencies and more control logic for dynamic shaders. The higher frame rate joined with additional free CPU time permit to achieve better interactivity and faster image convergence than with the software-only system.

The summary of our performance findings is:

- Demanding pixel shaders with tens of texture accesses per pixel are very fast, aside from the raw computational power of the GPUs.
- It is essential to process the greater amount of pixels with the lesser geometry to achieve maximum performance.
- Transferring vertex data on the GPU, such as RTV, as of this writing, is disappointingly slow to be fully useful.
- Lack of scatter writes on pixel shaders prevented us for managing all data on the GPU, and thus we had to copy back some information to the CPU.

5.1. Future work

Higher granularity of the pixel pipeline branching units will finally yield performance improvements on complex scenes at processes such as the pixel classification. This seems to be nearly possible on the newest generation of graphics hardware.

Different capabilities would allow us to implement a faster point projection. Scatter writes on the pixel shaders would allow us to implement the point projection and the predicted projection in a single pass, circumventing the bandwidth limitations. It would also permit us to implement other stages related to the management of the point cloud more efficiently.

Better RTV implementations would allow us to keep the point cloud as a texture on the GPU, and using an approach analogous to the age reset for updating the points. Using this technique with current hardware is slower than the dynamic VBO. More supported texture formats for

VTF, and faster performance, would make it an attractive way of update the point cloud data and project the points. The ability to modify data of the current vertex stream would allow us to eliminate the set image ID stage.

6. References

[1] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. ACM Trans. Graph., 22(3):631–640, 2003.

[2] Abhinav Dayal, Cliff Woolley, Benjamin Watson, and David P. Luebke. Adaptive frameless rendering. In Rendering Techniques, pages 265–275, 2005.

[3] Frederic P. Pighin, Dani Lischinski, and David Salesin. Progressive previewing of ray-traced images using image-plane discontinuity meshing. In Rendering Techniques ’97, pages 115–125, 1997.

[4] Timothy J. Purcell, Ian Buck,William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In SIGGRAPH ’02, pages 703–712, 2002.

[5] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In HWWS ’03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 41–50, 2003.

[6] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. ACM Trans. Graph., 22(3):521–526, 2003.

[7] M. Simmons and C. Séquin. Tapestry: A dynamic mesh-based display representation for interactive rendering, 2000.

[8] Miloslaw Smky, Shin-ichi Kinuwaki, Roman Durikovic, and Karol Myszkowski. Temporally coherent irradiance caching for high quality animation rendering. In Eurographics 2005, volume 24, 2005.

[9] Marc Stamminger, Joerg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective texturing. In Rendering Techniques 2000, pages 377–388, 2000.

[10] Laszlo Szirmay-Kalos, Barnabas Aszodi, Istvan Lazanyi, and Matyas Premecz. Approximate raytracing on the gpu with distance impostors. In Eurographics 2005, volume 24, 2005.

[11] Parag Tole, Fabio Pellacini, BruceWalter, and Donald P. Greenberg. Interactive global illumination in dynamic scenes. In SIGGRAPH ’02, pages 537–546, 2002.

[12] Bruce Walter, George Drettakis, and Donald P. Greenberg. Enhancing and optimizing the render cache. In Eurographics Workshop on Rendering, 2002.

[13] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In Rendering techniques ’99 (Proceedings of the 10th Eurographics Workshop on Rendering), volume 10, pages 235–246, Jun 1999.

I. Depth cull

The depth cull filter calculates an average depth from the valid simples in a pixel's 3x3 neighborhood. However, it needs to be shifted back slightly so that the edges that are exactly on the surfaces, such as the shadow ones, will not get culled, but at the same time those edges well behind the surface must not be drawn. The z-values used from the OpenGL depth buffer are not lineal, they depend on the position of the near and far clip planes. Therefore it is not possible to just shift back the depth values by a constant offset; instead that threshold must be calculated analytically.

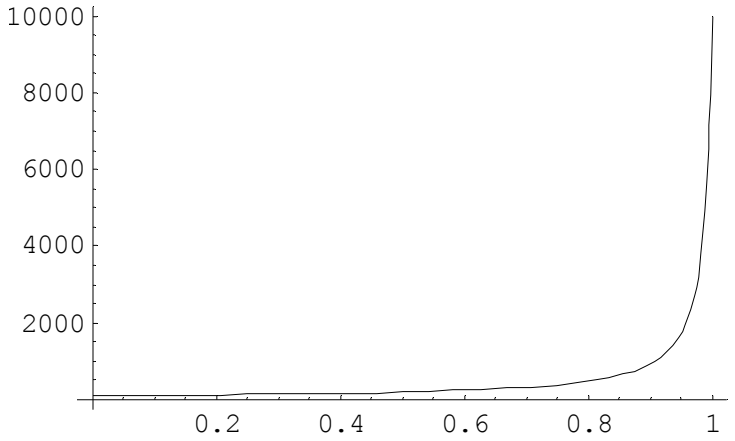


Figure 10: OpenGL normalized depth (*x*) *vs* the object space z-coordinates (*y*) for a near plane of 100 and a far plane of 10,000. As the objects are farther from the near plane the precision diminishes: 90% of the depth values are used for about 10% of the valid z range.

To get the new depth value, that will represent a constant displacement in object space we must find the object space value of the average depth found by the filter, move it back and then transform the result back into normalized depth values. For a z-coordinate z in object space, the depth value z_d using perspective projection, for a near plane n and a far plane f, is:

$$z_d = depth(z) = \frac{1}{2}\left(\frac{f+n}{f-n} - \frac{2fn}{z(f-n)}\right) + \frac{1}{2}$$

Conversely, the object space z value for a normalized depth z_d is:

$$z = depth^{-1}(z_d) = \frac{fn}{f - z_d(f-n)}$$

Then, the desired shifted-back filtered depth z'_d, displaced back by a factor c of the distance between the near and far planes from the average depth z_d is given by:

$$z'_d = depth\left(depth^{-1}\left(z_d\right) + c \cdot \left|f - n\right|\right) = \frac{f\left(-nz_d + c\left(f\left(z_d - 1\right) - nz_d\right)\right)}{-fn + c\left(f - n\right)\left(f\left(z_d - 1\right) - nz_d\right)}$$

This expression needs to be evaluated at every pixel, and expressed that way it would be very expensive to compute. As the terms *c,n,f* are constant at any given frame, we can express it in a more convenient way as:

$$z'_d = \frac{-cf^2 + f\left(cf - n - cn\right)z_d}{-cf\left(f - n\right) - fn + \left(cf\left(f - n\right) - c\left(f - n\right)n\right)z_d}$$

The previous expression might seem more complicated, but the values involving *c,n,f* can be computed on the CPU once per frame and then used on the GPU for every pixel. And as the pixel pipeline uses a SIMD model, we can express this conversion in terms of vector operations. Let us define the constant vectors v_1, v_2 calculated by the CPU and recorded as constant on the GPU registers:

$$v_1 = \left\langle f\left(cf - n - cn\right), cf\left(f - n\right) - c\left(f - n\right)n \right\rangle$$
$$v_2 = \left\langle -cf^2, -cf\left(f - n\right) - fn \right\rangle$$

Then the GPU can calculate the z'_d value with this vector operations, for a vector v_z defined for every pixel in terms of the average depth z_d:

$$v_z = \left\langle z_d, z_d \right\rangle$$
$$v_t = v_1 * v_z + v_2$$
$$z_d = \frac{v_t.x}{v_t.y}$$

Therefore, for each average depth the GPU will execute only two operations, one vector multiply-and-add, and a scalar division.

II. Subpixel information calculation

As each point in the point cloud is projected to exactly one pixel, it is possible to write the relative position of the point inside the pixel using a vertex shader with the window coordinates of the point. Then the pixel shader will record the information directly as received from the vertex program.

At the vertex shader the only additional information needed is the width and height of the target image. Then the subpixel information is obtained with the normalized eye space coordinates of the vertex. Let M_{MVP} be the concatenated model-view/projection matrix, and p is the homogeneous position of the point in world space. The eye coordinates p_e are:

$$p_e = (M_{MVP})p$$

After the perspective division we obtain the three-component normalized eye space vector p_h. Each component on this vector lies in the range *[-1,1]* if they are inside the view frustum:

$$p_h = \frac{p_e.xzy}{p_e.w}$$

Once in normalized eye space we only need to complete the transformation to the window coordinates (x_w,y_w):

$$x_w = \frac{width}{2}(p_h.x + 1)$$
$$y_w = \frac{height}{2}(p_h.y + 1)$$

Finally, the mantissa (the fractional part) of each coordinate gives the relative position inside the pixel, from the bottom left corner to the top right corner of the pixel: a *(0,0)* value means that the sample is at the bottom left corner, *(0.5, 0.5)* is the center of the pixel and *(1,1)* would be the top right corner.

III. Setting the Image ID

To set the image ID (the index of the pixel to which a point was projected) the easiest option would be storing that value when the point is being projected. However, with the streamlined architecture of the GPU this is impossible; hence a separate rendering pass is required. The methodology is similar to that of the point update stage: for each pixel in the age – image ID texture to be updated, project a point to that position and write the new value.

Potentially, each pixel of the point ID texture implies that the point which was projected to that pixel must be updated and have its image ID set to the index of that pixel. Not all the pixels have a point mapped to them, but must of them will.

Each RGBA8 pixel of the point ID texture will be interpreted as two short coordinates. This way the point ID will be passed encoded with the position. The texture is read into a Pixel Buffer Object (PBO), which attachment point is then changed in order to use it as a vertex array. This is a way of implementing Render To Vertex array (RTV).

In a vertex shader, the position will be interpreted as the point ID, directing the vertex to the proper position in the age – image ID texture. When the vertices are passed, they are read in little endian format. For each pixel with 8 bit unsigned bytes values, RGBA (stored in that order), the x,y coordinates are:

$$x = (\text{short})\ (\ (G \ll 8)\ |\ (R)\ ) = (\text{short})\ ((\ G * 2^8\ ) + R\ )$$
$$y = (\text{short})\ (\ (A \ll 8)\ |\ (B)\ ) = (\text{short})\ ((\ A * 2^8\ ) + B\ )$$

And the point ID is:

$$\text{pointID} = (A \ll 16)\ |\ (B \ll 8)\ |\ (G) = (A * 2^{16}) + (B * 2^8) + G$$

When two bytes are interpreted as signed shorts, any value beyond 0x7FFF is considered to be a negative number, interpreted in two's complement notation. Also, the vertex processor receives that number as a floating point value, and is unable to perform bit operations on it, so it must be transformed using different methods to get back the desired values.

To convert back a signed number n of m bits in two's complement to the number the same bits would represent if it were interpreted as an unsigned value, it is simply required to add 2^m to n: 0xFF ~ -1; $1 + 2^8 = 255$.

All pixels of the point ID texture have a value for the flags field, ergo for all x-coordinates, R > 0 is true, so:

$$G = \left\lfloor \frac{x}{256} \right\rfloor = \left\lfloor \frac{G \times 2^8 + R}{2^8} \right\rfloor = \left\lfloor G + \frac{R}{2^8} \right\rfloor = G, G \in [-128,255], x \in [-2^{15}, 2^{15} - 1]$$

Thus the point ID p can be reconstructed from the encoded *x,y* coordinates with the formula:

$$p = \left\lfloor \frac{x}{2^8} \right\rfloor + 2^8 \cdot y + bias$$

$$bias = \begin{cases} \frac{x}{2^8} < 0,2^8 \\ \frac{x}{2^8} \geq 0,0 \end{cases}$$

The *bias* converts back only the G component back from a negative value to a positive one, because the R gets eliminated in the process. There is no need to add an additional bias for *y* values, unless A > 127, which would mean that there are 2^{23} or more pixels. That number corresponds to an image larger than *2884* x *2884*.

With the point ID, the vertex shader calculates the appropriate *x,y* homogeneous coordinates of the image ID texture into which it must be projected. Analogous to the point ID value in the point projection, the *i*-th pointID being projected in this staged was mapped to the *i*-th pixel of the image, therefore the image ID is calculated in advance and stored in a VBO. Only 24 bits are used for the image ID, whose LSB is stored in the Red channel and the MSB is in the Blue channel.

To perform the actual drawing on the texture, it is masked so that the age, which is in the Red channel, does not get modified in the process. Then the buffer is cleared with the color (0, 0xFF, 0xFF, 0x7F). The points that get this image ID are those that were not projected into the image. The highest bit in the Alpha channel is left unused here to set up the penalization flag employed when aging the points.

A pixel shader reads the pre-computed image ID encoded as RGB color, and copies it to the GBA channels, only displacing it one position to the right in order to leave room for the age in the Red channel. Finally, in the texture the age is in the Red channel, the image ID's LSB is in the Green channel and its MSB is in the Alpha channel. This method, although complicated and not straight forward, has the advantages of not requiring additional custom data, and all the processing remains in the GPU at all times. No information is copied from the GPU to the CPU or vice versa.

IV. Parsing results

The output of the RTGI included timings distributed in multiple rows grouped by frames; however the data needed to be summarized in columns grouped by categories to feed the data into spreadsheet software and analyze the results. To perform that transformation we used an *awk* program. Here is the program listing for the parser of the GPU version output, to be used as future reference.

```
# Small awk program to parse the benchmarking output to a format
# suitable for MS Excel
#
# Filename: parser.awk
# Usage: awk -f parser.awk <src_file>
#
# Edgar Velázquez-Armendáriz (edgar@graphics.cornell.edu)
# August 10th 2005
#
BEGIN { count = 0; countAlfa = 0; countBeta = 0; }

/Sil cumulative/          { silCumulative[++countAlfa]=$4; }
/Hard shadows/            { hardShadows[countAlfa]=$4; }


/Point predict/           { pointPred[++countBeta]=$5; }
/Point update\./          { pointUpdate[countBeta]=$5; }
/Point projection avg/    { pointProj[countBeta]=$5; }
/Point depth cull avg/    { depthCull[countBeta]=$6; }
/Edge initialize/         { edgeInit[countBeta]=$5; }
/Edge rasterize/          { edgeRast[countBeta]=$5; }
/Edge reconstruct/        { edgeReconstruct[countBeta]=$5; }
/Interpolate/             { interpolate[countBeta]=$4; }
/Anti alias/              { antAlias[countBeta]=$5; }
/Request samples/         { requestSamples[countBeta]=$5; }
/Frame total/             { frameTotal[countBeta]=$5; }


/Point update RC/         { pointUpdateRC[++count]=$6; }
/Point setup RC/          { pointSetupRC[count]=$6; }
/Point projection RC/     { pointProjRC[count]=$6; }
/Predicted projection/    { predProjRC[count]=$6; }
/- Read pred. proj/       { readPredProjRC[count]=$7; }
/Point depth cull RC/     { depthCullRC[count]=$7; }
/- Read pointID/          { readPointid[count]=$6; }
/Age points RC/           { agePointsRC[count]=$6; }
/Set ImageID RC/          { setImageId[count]=$6; }
/Readback ImageID-age/    { readbackAge[count]=$5; }


/Z Update/          { zUpdate[count]=$5; }
/Image Copy/        { imageCopy[count]=$5; }
/Edge Buffer/       { edgeBuffer[count]=$5; }
/Silhouette/        { silhouette[count]=$4; }
/- Detection/       { silhDetection[count]=$5; }
/Render Input/      { renderInput[count]=$5; }
/Edge Raster/       { edgeRaster[count]=$5; }
/Pixel Class/       { pixelClass[count]=$5; }
```

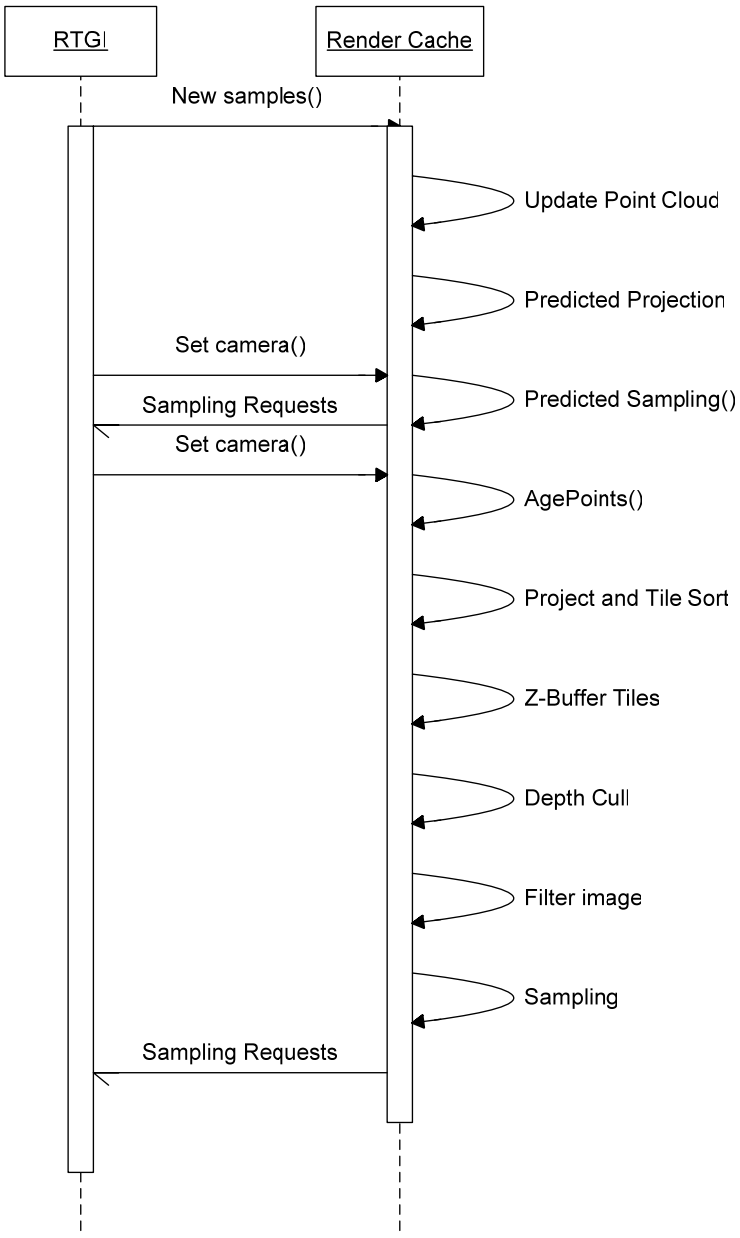
```
/Point Cull/        { pointCull[count]=$5; }
/Reachability/      { if ($4 > 0) reachability[count]=$4; }
/- Copy/            { copy[count]=$5; }
/Interpolation/     { interpolation[count]=$4; }
/AntiAliasing/      { antiAliasing[count]=$4; }
/- Readback/        { readback[count]=$5; }
/Tot Rendering/     { totRendering[count]=$5; }
/Total GPU/         { totGPU[count]=$5; }

END {
       printf("Count: %d\nFile parsed: %s\n", count, FILENAME);

       # Prints everything!

       printf("\n%-18s\t", "Sil cumulative:");
       for (i=1; i<=countAlfa; i++) printf("%f\t", silCumulative[i]);

       printf("\n%-18s\t", "Hard shadows:");
       for (i=1; i<=countAlfa; i++) printf("%f\t", hardShadows[i]);



       printf("\n------------------------------\n%-18s\t", "Point update:");
       for (i=1; i<=countBeta; i++) printf("%f\t", pointUpdate[i]);

       printf("\n%-18s\t", "Point projection:");
       for (i=1; i<=countBeta; i++) printf("%f\t", pointProj[i]);

       printf("\n%-18s\t", "Point predict:");
       for (i=1; i<=countBeta; i++) printf("%f\t", pointPred[i]);

       printf("\n%-18s\t", "Point depth cull:");
       for (i=1; i<=countBeta; i++) printf("%f\t", depthCull[i]);

       printf("\n------------------------------\n%-18s\t", "Edge initialize:");
       for (i=1; i<=countBeta; i++) printf("%f\t", edgeInit[i]);

       printf("\n%-18s\t", "Edge rasterize:");
       for (i=1; i<=countBeta; i++) printf("%f\t", edgeRast[i]);

       printf("\n%-18s\t", "Edge reconstruct:");
       for (i=1; i<=countBeta; i++) printf("%f\t", edgeReconstruct[i]);

       printf("\n%-18s\t", "Reachability RC:");
       for (i=1; i<=countBeta; i++) printf("%f\t", reachRC[i]);

       printf("\n%-18s\t", "Interpolate:");
       for (i=1; i<=countBeta; i++) printf("%f\t", interpolate[i]);

       printf("\n%-18s\t", "Anti alias:");
       for (i=1; i<=countBeta; i++) printf("%f\t", antAlias[i]);

       printf("\n%-18s\t", "Request samples:");
       for (i=1; i<=countBeta; i++) printf("%f\t", requestSamples[i]);

       printf("\n%-18s\t", "Frame total:");
       for (i=1; i<=countBeta; i++) printf("%f\t", frameTotal[i]);



       printf("\n------------------------------\n%-18s\t", "Point update RC:");
```

```
for (i=1; i<=count; i++) printf("%f\t", pointUpdateRC[i]);

printf("\n%-18s\t", "Point setup RC:");
for (i=1; i<=count; i++) printf("%f\t", pointSetupRC[i]);

printf("\n%-18s\t", "Point projection R:");
for (i=1; i<=count; i++) printf("%f\t", pointProjRC[i]);

printf("\n%-18s\t", "Predicted projection RC:");
for (i=1; i<=count; i++) printf("%f\t", predProjRC[i]);

printf("\n%-18s\t", " - Read pred. proj.");
for (i=1; i<=count; i++) printf("%f\t", readPredProjRC[i]);

printf("\n%-18s\t", "Point depth cull R:");
for (i=1; i<=count; i++) printf("%f\t", depthCullRC[i]);

printf("\n%-18s\t", " - Read pointID:");
for (i=1; i<=count; i++) printf("%f\t", readPointid[i]);

printf("\n%-18s\t", "Set ImageID RC:");
for (i=1; i<=count; i++) printf("%f\t", setImageId[i]);

printf("\n%-18s\t", "Age points RC:");
for (i=1; i<=count; i++) printf("%f\t", agePointsRC[i]);

printf("\n%-18s\t", "Readback ImageID-age:");
for (i=1; i<=count; i++) printf("%f\t", readbackAge[i]);



printf("\n------------------------------\n%-18s\t", "Z Update:");
for (i=1; i<=count; i++) printf("%f\t", zUpdate[i]);

printf("\n%-18s\t", "Image Copy:");
for (i=1; i<=count; i++) printf("%f\t", imageCopy[i]);

printf("\n%-18s\t", "Edge Buffer:");
for (i=1; i<=count; i++) printf("%f\t", edgeBuffer[i]);

printf("\n------------------------------\n%-18s\t", "Silhouette:");
for (i=1; i<=count; i++) printf("%f\t", silhouette[i]);

printf("\n%-18s\t", " - Detection:");
for (i=1; i<=count; i++) printf("%f\t", silhDetection[i]);

printf("\n%-18s\t", "Render Input:");
for (i=1; i<=count; i++) printf("%f\t", renderInput[i]);

printf("\n%-18s\t", "Edge Raster:");
for (i=1; i<=count; i++) printf("%f\t", edgeRaster[i]);

printf("\n%-18s\t", "Pixel Class:");
for (i=1; i<=count; i++) printf("%f\t", pixelClass[i]);

printf("\n%-18s\t", "Point Cull:");
for (i=1; i<=count; i++) printf("%f\t", pointCull[i]);

printf("\n%-18s\t", "Reachability:");
for (i=1; i<=count; i++) printf("%f\t", reachability[i]);
```

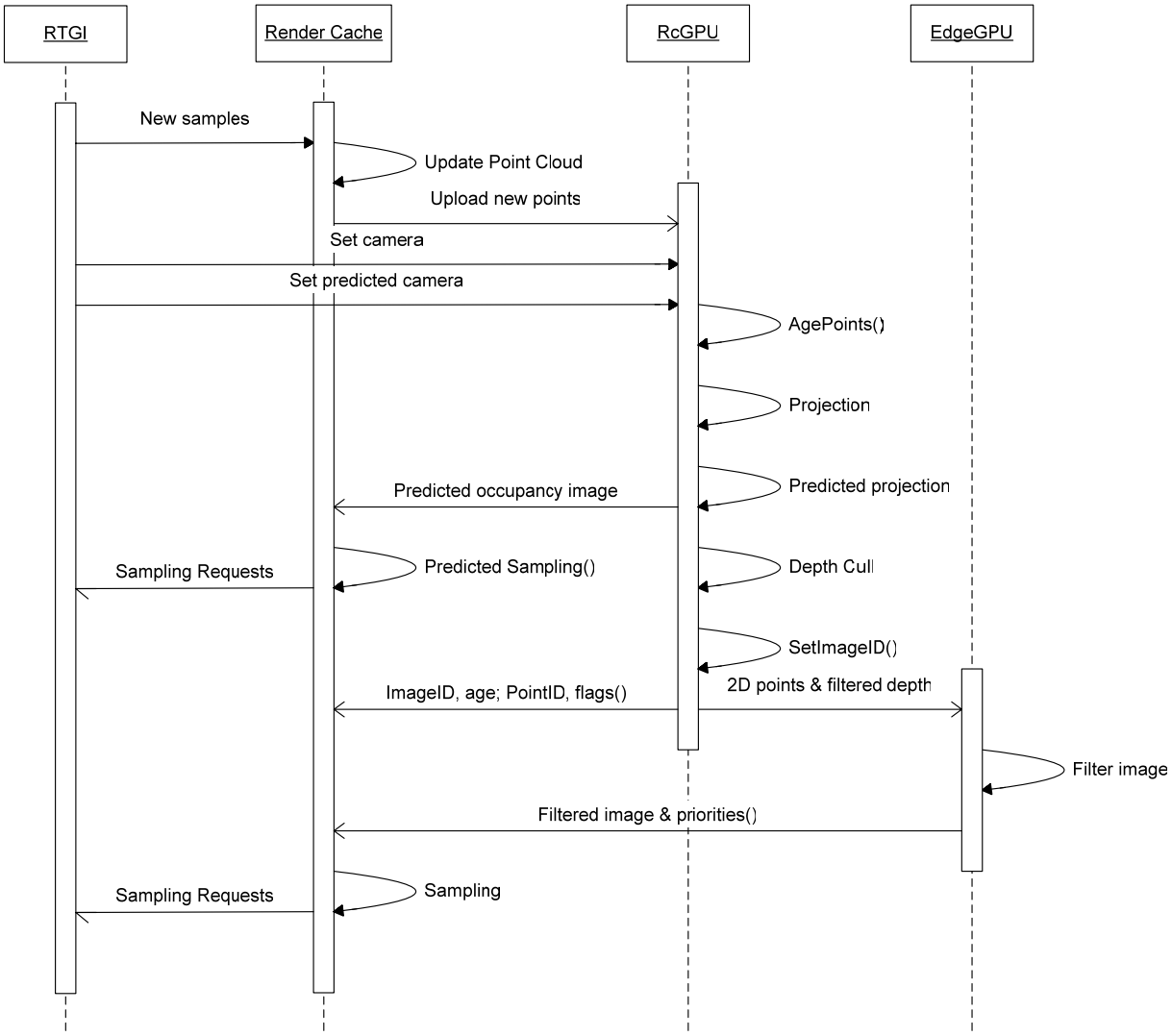
```
        printf("\n%-18s\t", " - Copy:");
        for (i=1; i<=count; i++) printf("%f\t", copy[i]);

        printf("\n%-18s\t", "Interpolation:");
        for (i=1; i<=count; i++) printf("%f\t", interpolation[i]);

        printf("\n%-18s\t", "AntiAliasing:");
        for (i=1; i<=count; i++) printf("%f\t", antiAliasing[i]);

        printf("\n%-18s\t", " - Readback:");
        for (i=1; i<=count; i++) printf("%f\t", readback[i]);

        printf("\n------------------------------\n%-18s\t", "Tot Rendering:");
        for (i=1; i<=count; i++) printf("%f\t", totRendering[i]);

        printf("\n%-18s\t", "Total GPU:");
        for (i=1; i<=count; i++) printf("%f\t", totGPU[i]);


        printf("\n\n");


}
```

V. System's UML and DFD Diagrams

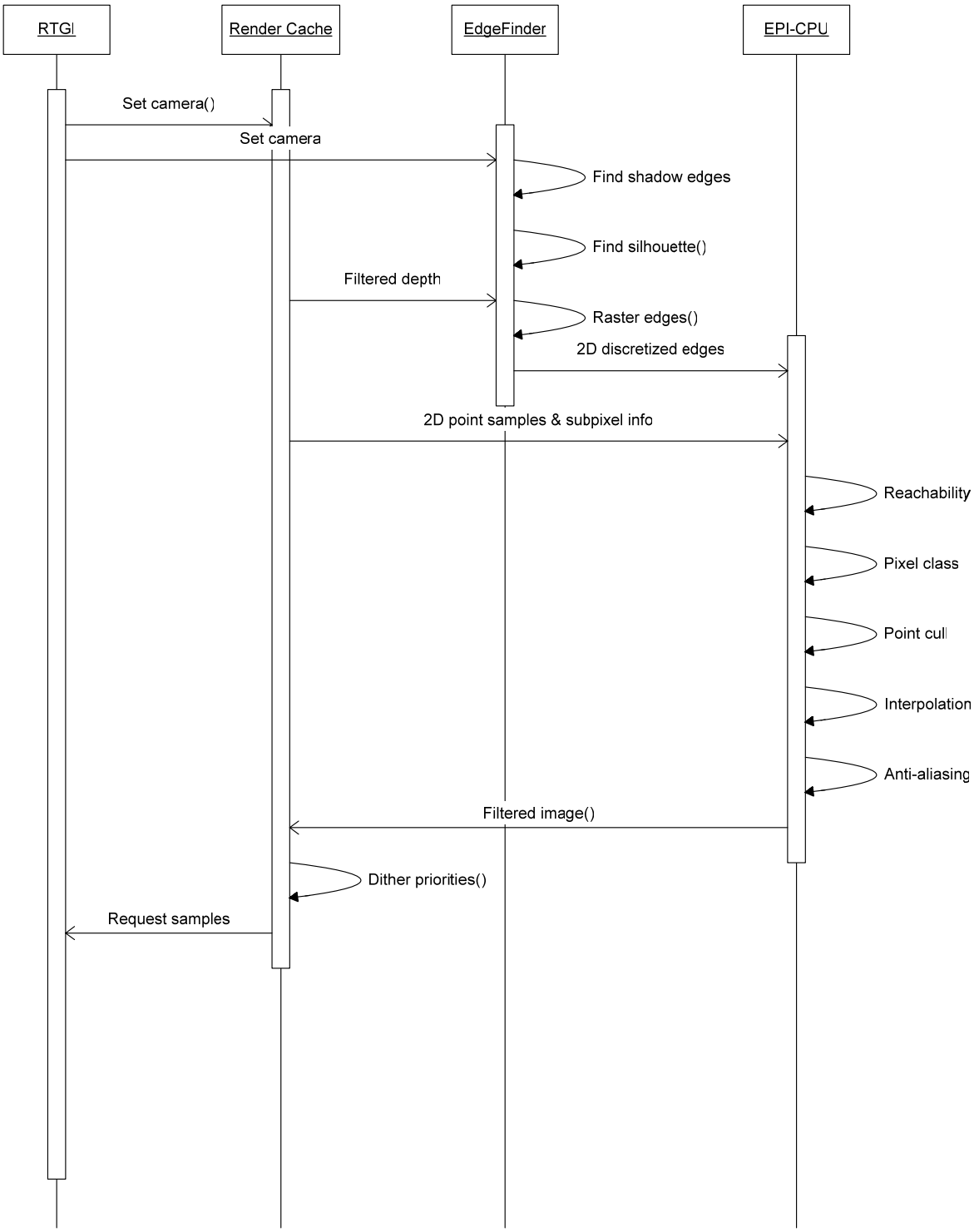
Original system



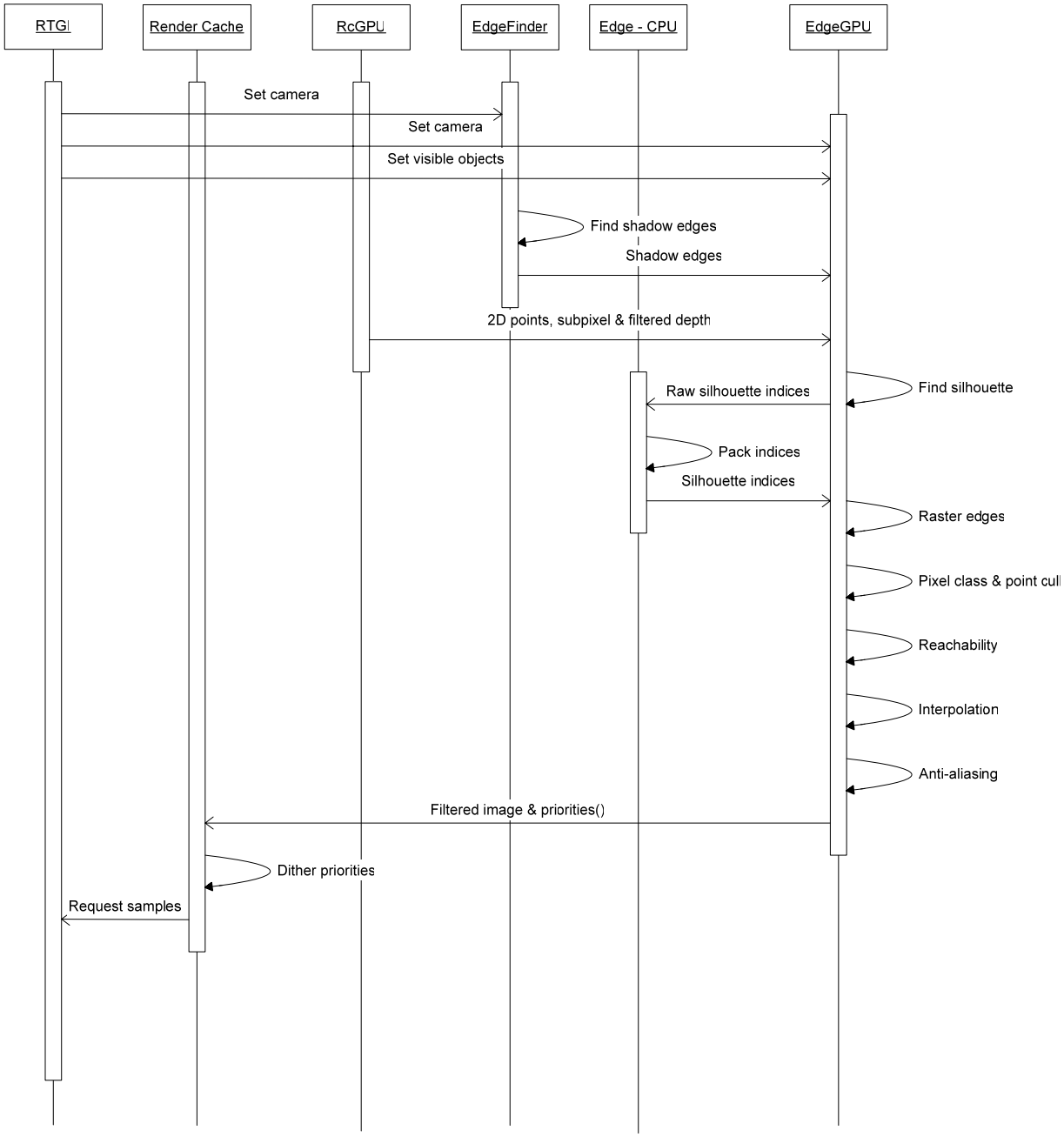
GPU system


RTGI
Render Cache
RcGPU
EdgeGPU
New samples
Update Point Cloud
Upload new points
Set camera
Set predicted camera
AgePoints()
Projection
Predicted projection
Predicted occupancy image
Predicted Sampling()
Depth Cull
Sampling Requests
SetImageID()
ImageID, age; PointID, flags()
2D points & filtered depth
Filter image
Filtered image & priorities()
Sampling
Sampling Requests

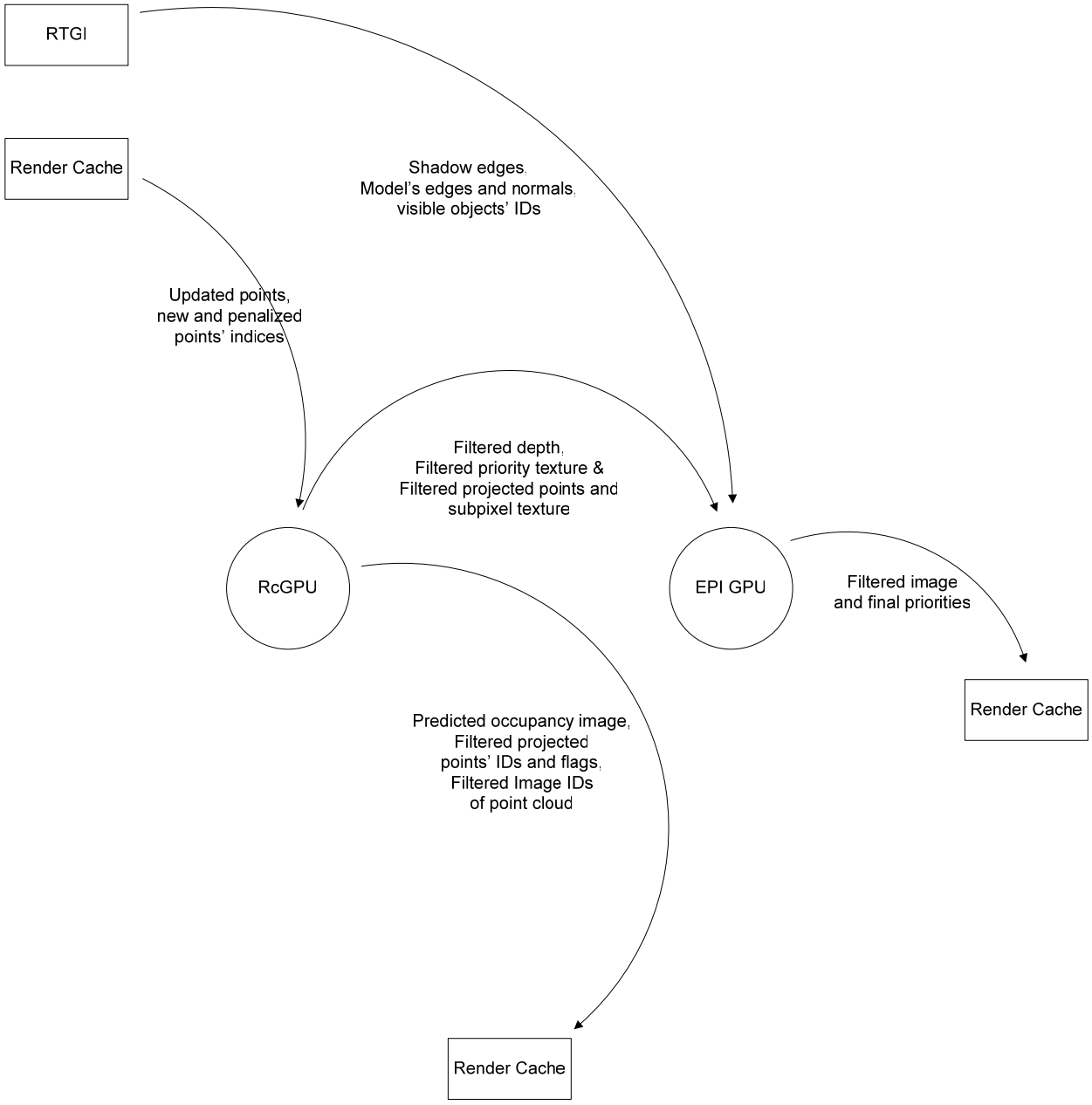
EPI Filter


RTGI
Render Cache
EdgeFinder
EPI-CPU
Set camera()
Set camera
Find shadow edges
Find silhouette()
Filtered depth
Raster edges()
2D discretized edges
2D point samples & subpixel info
Reachability
Pixel class
Point cull
Interpolation
Anti-aliasing
Filtered image()
Dither priorities()
Request samples

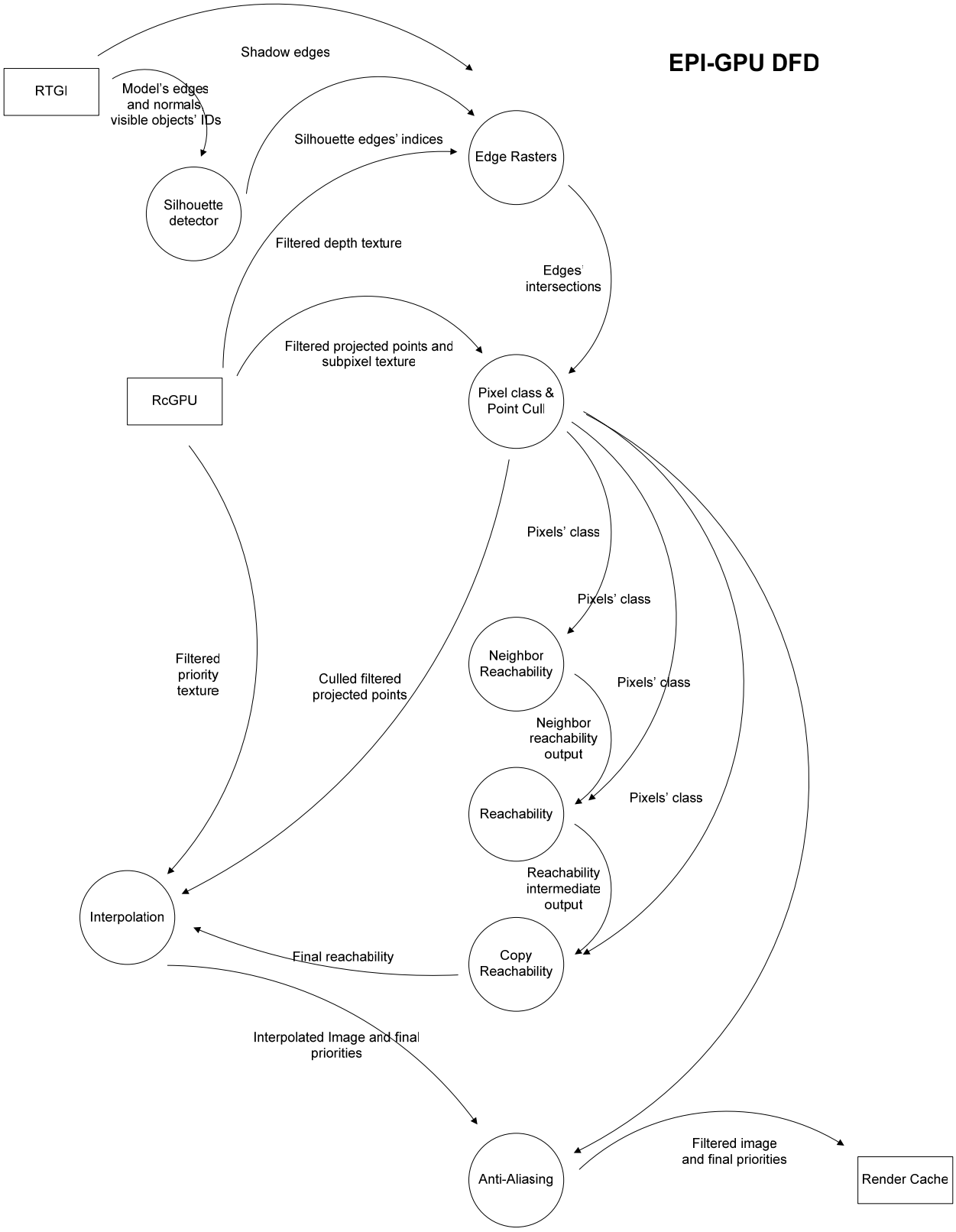
EPI-GPU Filter



GPU System DFD



GPU Point Projector (RcGPU) DFD


EPI-GPU DFD
RTGI
Shadow edges
Model's edges and normals visible objects' IDs
Silhouette detector
Silhouette edges' indices
Edge Rasters
Filtered depth texture
Edges' intersections
RcGPU
Filtered projected points and subpixel texture
Pixel class & Point Cull
Pixels' class
Pixels' class
Neighbor Reachability
Filtered priority texture
Culled filtered projected points
Pixels' class
Neighbor reachability output
Reachability
Pixels' class
Reachability intermediate output
Interpolation
Final reachability
Copy Reachability
Interpolated Image and final priorities
Anti-Aliasing
Filtered image and final priorities
Render Cache